Is Your Graph Algorithm Eligible for Nondeterministic Execution?

Zhiyuan Shao, Lin Hou, Yan Ai, Yu Zhang and Hai Jin
Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
{zyshao,hjin}@hust.edu.cn

Abstract—Graph algorithms are used to implement data mining tasks on graph data-sets. Besides conducting the algorithms by the default deterministic manner, some graph processing frameworks, especially those supporting asynchronous execution model, provide interfaces for the algorithms to be executed in nondeterministic manner, which can improve the scalability and performance of the algorithm’s executions. However, is the graph algorithm eligible for nondeterministic execution, and will the execution produce expected results? The literature gives few answers to these questions.

In this paper, we study the nondeterministic execution of graph algorithms by considering the scenario where data dependences happen in the edges in graph processing frameworks that employ asynchronous execution model. Our study reveals that only by guaranteeing the atomicity of individual reads and writes, some algorithms (e.g., graph traversal algorithms) can converge by recovering from corrupted intermediate results with nondeterministic execution, and thus tolerate even write-write conflicts, while some other algorithms (e.g., fixed point iteration algorithms) can converge but tolerate only read-write conflicts. By conducting graph algorithms on real-world graphs in GraphChi, and comparing their performances and results with deterministic executions, we find that their performance gains are generally scalable to the available processors with nondeterministic executions, and the results at convergence of fixed point iteration algorithms from nondeterministic executions exhibit larger variances from one run to another than their deterministic executions.

Keywords—graph processing; nondeterministic execution; data race; parallel programming; scheduling

I. INTRODUCTION

As it is rational to use graphs to model the real-world problems, and solve these problems by graph algorithms, graph processing has attracted more and more attention from academia and industry, and becomes one of the key research problems in the big-data era. Within the past several years, lots of graph processing systems, such as Pregel [1], GraphLab [2, 3], GraphChi [4], PowerGraph [5], Galois [6], Ligra [7] and many others, had been built to provide general frameworks to facilitate the users to program and conduct their algorithms on graphs data-sets.

By graph topology, graph processing can be considered as conducting local update tasks (updates for short) on the vertices, and the edges are generally used to pass intermediate results (i.e., messages) between the updates. Since today’s real-world graphs have millions, even billions of vertices, processing such graph data-sets needs to handle huge number of updates. Two categories of scheduling algorithms are generally employed to do this job [6]: autonomous scheduling and coordinated scheduling. By autonomous scheduling, a graph algorithm is allowed to define the execution path of the updates so as to accelerate its convergence. By coordinated scheduling, the graph algorithm has no control over the execution path of updates, and the scheduler organizes the updates by multiple iterations (i.e., rounds): at each iteration a set of updates is chosen, and each of chosen updates is executed exactly once. Coordinated scheduling is more widely supported than autonomous scheduling in the graph processing frameworks due to its simplicity.

Two execution models, i.e., the synchronous model and asynchronous model, exist in the systems that employ coordinated scheduling. Synchronous model is also named as Bulk Synchronous Parallel (BSP) model [8], where the effectiveness of the updates are postponed and visible at the beginning of next iteration. Asynchronous model is also named as Gauss-Seidel (GS) model, where the updates to objects (i.e., vertices and edges) become effective immediately, and their results are visible to the following updates. By synchronous model, the postponed effectiveness of results exempts data dependences among the updates of the same iteration, and thus improves the parallelism of execution. However, by solving the same algorithm on the same graph data-set, synchronous model generally needs to conduct more iterations than asynchronous model, which greatly counteracts the benefits gained from the parallelism [9]. Conversely, in asynchronous model, although the immediate effectiveness of updates reduces the number of iterations required by the algorithms for convergence, the updates have to consider the data dependences among themselves when executed concurrently. Methods (e.g., set scheduler of GraphLab [2], external deterministic scheduler of GraphChi [4], chromatic scheduler [10] and deterministic interference graph scheduler in Galois [11]) are proposed to define deterministic execution paths for the chosen updates so as to prevent data races (i.e., multiple operations are conducted concurrently to the same memory location, and at least one of them is a write [20]) as well as improve parallelism.
However, plotting the execution path at the beginning of each iteration incurs huge time and space overheads.

Although executing the updates nondeterministically in asynchronous model can hopefully avoid the overheads introduced by plotting execution paths for deterministic executions, and naturally improve the scalability of computing, we find few literatures explaining this execution manner.

In this paper, we study the nondeterministic execution of the graph algorithms in the scenario where there are data dependences on the edges. Our study reveals that only by guaranteeing the atomicity of individual reads and writes of their update tasks, most graph algorithms can converge by nondeterministic executions: by giving two sufficient conditions, we prove that in the considered scenario, some algorithms (e.g., graph traversal algorithms) can converge by recovering from corrupted intermediate results by nondeterministic executions, and thus tolerate even write-write conflicts, while some other algorithms (e.g., fixed point iteration algorithms) can converge with the presence of read-write conflicts.

This paper makes following contributions:

- Studies the nondeterministic execution of graph algorithms in systems that employ coordinated scheduling of asynchronous model, and gives two sufficient conditions to guide the programmers for the adoption of nondeterministic execution to their algorithms;
- Proposes three applicable methods of guaranteeing the atomicity of individual read and write in the updates of a graph algorithm, and shows their corresponding effects to the algorithms’ performances of nondeterministic executions when processing real-world graphs;
- Evaluates the results produced by the nondeterministic executions of graph algorithms, and points out the high variances in results of fixed point iteration graph algorithms from one run to another.

The rest of this paper is organized as follows: Section II presents the system model. Section III discusses the constraints to the updates. Section IV discusses the sufficient conditions to the convergence of nondeterministically executed graph algorithms. Section V conducts experiments to evaluate the performances and results of nondeterministically executed graph algorithms by processing real-world graphs. Section VI presents a brief survey on the related works. Section VII concludes the paper and presents the future works.

## II. System model

We denote the graph in discussion as $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges. Each edge is directed (undirected edge can be represent by two edges with opposite directions) and denoted as $(u \rightarrow v)$, where $u, v \in V$. When there raises no confusion, $V$ and $E$ will also be used to represent the number of vertices and edges respectively. We denote the data associated with vertex $v$ as $D_v$, and the data associated with edge $(u \rightarrow v)$ as $D_{u \rightarrow v}$.

An unique label is defined and associated with each vertex $v \in V$, such that no two individual vertices in graph $G$ share the same label. We denote the label value of $v$ as $L_v$. In order to simplify discussion, the serial number of $v$ is used as its label value, i.e., $L_v \in \{0, ..., V - 1\}$.

**Update function:** We denote the graph algorithm in discussion as $A$. By employing the vertex-centric programming model $[1]$, $A$ defines an update function $f(v) \forall v \in V$. The scope of $f(v)$ is defined as all the adjacent objects of $v$. During execution, $f(v)$ can only read or write the data of the objects within its scope. As in this paper we consider only data-dependences on edges, we further shrink the scope of an update function to its corresponding vertex and all its incident edges (i.e., pull mode $[6]$). As graph algorithms can generally be organized as multiple Gather-Compute-Scatter iterations $[12]$, using $E_r$ to denoted the set of incident edges of $v$, $E_c$ and $E_w$, which are subsets of $E_r$, to denote the set of edges of that $f(v)$ reads during the gather phase and writes during the scatter phase respectively. $f(v)$ can be modeled in Algorithm 1. The two subsets of $E_r$ and $E_w$ can be arbitrarily taken from $E_r$, and some algorithms may optionally read the edge before writing during the scatter phase.

**Algorithm 1 Update function of $f(v)$**.

1: Gather:
2: for all $e \in E_r$ do
3: Read the value of edge $e$
4: end for
5: Compute:
6: Compute the result
7: Scatter:
8: for all $e \in E_w$ do
9: if $e$ satisfy some criteria{for some algorithms} then
10: Write the result to edge $e$
11: end if
12: end for

Since update functions are encapsulated as update tasks, we will use these two terms interchangeably, and use updates for short when there raises no confusion.

**Scheduling:** We consider conducting graph algorithm $A$ on $G$ in a shared memory architecture machine $M$, which has $P$ ($P \geq 1$) processors, and enough memory space to load the whole graph during computing. $T$ threads are invoked during the computation, and we assume $T = P$ in this paper to simplify discussions. Although by asynchronous execution model, there will be no explicit synchronization points, i.e., barriers, our system model still includes the synchronization points. That is, we consider the synchronous implementation of asynchronous model in this paper. Since
in some memory models (e.g., Java memory model [14]), without the barriers, there will be no guarantee for a read to obtain the latest value of an object. With the barriers, the objects will commit to one predictable value, which greatly simplifies our discussions. Besides, as proven by GRACE [13], such implementation has comparable runtime to those of pure asynchronous model.

Therefore, the execution of \( A \) in our system model consists of \( N \) rounds of iterations, denoted as \( I_0, I_1, ..., I_{(N-1)} \). \( I_0 \) is the initial state, and after the execution of \( I_{(N-1)} \), \( A \) converges. At the beginning of an arbitrary iteration (say the \( n^{th} \) iteration, where \( 0 \leq n < (N-1) \)), a set of updates are selected to execute. We denote the set of corresponding vertices of the chosen updates at the \( n^{th} \) iteration as \( S_n \), such that \( S_n \subseteq V \). During the \( n^{th} \) iteration, the execution of update tasks in \( S_n \) will (possibly) generate new tasks, which are to be executed in the \( (n+1)^{th} \) iteration. Our system model only places one simple rule regarding to the behaviors of task generating: if during the \( n^{th} \) iteration, \( f(v) \) updates one of \( v \)'s incident edges, say \( v \to u \) or \( v \leftarrow u \), it must add \( u \) to \( S_{n+1} \).

Our system model dispatches the chosen updates of an iteration among the participating threads by round-robin fashion, as illustrated in Fig. 1 (here we assume \( V \) is exactly divisible by \( T \), and \( S_n = V \)). This fashion actually complies with the method of the static scheduling by OpenMP runtime system [15]. Each participating thread executes its assigned updates according to the small-label-first rule, i.e., first conducts the update task whose corresponding vertex has the smallest label value. After having finished all assigned tasks of one iteration, a participating thread has to wait other threads before the start of the next iteration.

Orders among updates: We first define an absolute scheduling order among the updates of the same iteration, and denote it for update \( f(v) \) as \( \pi(v) \). For the situation illustrated in Fig. 1, we have \( \pi(v) = L_v \% (V/P) \). Considering two vertices \( v \) and \( u \), if \( \pi(v) < \pi(u) \), \( f(v) \) will be scheduled before \( f(u) \), and vice versa (we assume that each update task has the same execution time). Based on the definition of the absolute scheduling order, we further define partial orders, denoted by operators \(<, \succ \) and \( \parallel \), between two update functions as the follows.

**Definition 1.** We say \( f(v) \prec f(u) \), if one of following two conditions satisfy:
1) \( f(v) \) and \( f(u) \) are scheduled in the same thread, and \( \pi(v) < \pi(u) \);
2) \( f(v) \) and \( f(u) \) are scheduled in different threads, and \( (\pi(u) - \pi(v)) \geq d \).

**Definition 2.** We say \( f(v) \succ f(u) \), if one of following two conditions satisfy:
1) \( f(v) \) and \( f(u) \) are scheduled in the same thread, and \( \pi(v) > \pi(u) \);
2) \( f(v) \) and \( f(u) \) are scheduled in different threads, and \( (\pi(v) - \pi(u)) \geq d \).

**Definition 3.** We say \( f(v) \parallel f(u) \), if both of the following two conditions satisfy:
1) \( f(v) \) and \( f(u) \) are scheduled in different threads;
2) \( |\pi(v) - \pi(u)| < d \).

where \( d \) is a constant parameter, whose value choosing depends on the implementation details of the machine (i.e., the cache coherent protocol among processors, e.g., [16]). \( d \) denotes the time (measured by the number of updates) for the result of an update to propagate from one thread to another. Therefore, the implication of the definitions are:
- \( f(v) \prec f(u) \): \( f(u) \) can use the results from \( f(v) \), but \( f(v) \) cannot use the results from \( f(u) \).
- \( f(v) \succ f(u) \): \( f(v) \) can use the results from \( f(u) \), but \( f(u) \) cannot use the results from \( f(v) \).
- \( f(v) \parallel f(u) \): both \( f(u) \) and \( f(v) \) cannot use the results from each other.

Note that Fig. 1 depicts a simple scenario that facilitates our discussions. In real systems, as the scheduling mechanism of underlying runtime system may change, and the configuration of the machine may also change (e.g. increase the number of threads or change the cache coherence protocol), there will be no predefined order of execution of the updates when they are conducted nondeterministically.

III. ATOMICITY GUARANTEING

Since vertices are connected with edges, nondeterministic execution of the update tasks inevitably results in dangerous data races [17, 18] when concurrently accessing their common edges. In order to prevent such data races, the method of guaranteeing the atomicity of operations in the update function is widely employed. However, there are different choices on the granularity of such atomicity guaranteeing. For example, in Ligra [7] the update function works in push mode, and use atomic compare-and-swap
(CAS) to guarantee the atomicity, i.e., the granularity is the whole update function. Obviously, larger granularity inevitably results higher synchronization overhead and worse performance.

In our model, we only enforce the atomicity guarantee to the individual reads and writes, i.e., the minimal granularity for atomicity guaranteeing. That is, the update function only needs to guarantee that its individual operation (read or write) should not be interrupted when accessing (reading or writing) to a memory location. By this atomicity guaranteeing, data races are eliminated, and each read or write will not raise unexpected result as discussed in [19]. Therefore, we have following lemmas (the proof are rather straightforward, and thus omitted due to space constraint).

**Lemma 1.** There are two adjacent vertices \( v \) and \( u \), connected by edge \((v \rightarrow u)\) whose original value is \( D'_{(v \rightarrow u)} \), and during one iteration, \( f(v) \) and \( f(u) \) are conducted concurrently. One update (say \( f(v) \)) will read edge \((v \rightarrow u)\), while the other update (say \( f(u) \)) will write its computed result \((D''_{(v \rightarrow u)})\) to edge \((v \rightarrow u)\). In this scenario, the edge transmits either \( D'_{(v \rightarrow u)} \) or \( D''_{(v \rightarrow u)} \) to \( f(v) \), and its data will be \( D''_{(v \rightarrow u)} \) at the end of the iteration.

**Lemma 2.** There are two adjacent vertices \( v \) and \( u \), connected by edge \((v \rightarrow u)\), and during one iteration, \( f(v) \) and \( f(u) \) are conducted concurrently. One update (say \( f(v) \)) will write its computed result \((D'_{(v \rightarrow u)})\) and the other update will write its computed result \((D''_{(v \rightarrow u)})\) to the edge. At the end of the iteration, the data of edge \((v \rightarrow u)\) is either \( D'_{(v \rightarrow u)} \) or \( D''_{(v \rightarrow u)} \).

Moreover, since by guaranteeing the atomicity of the individual read and write, there will be explicit or implicit happen-before relationships [20] between two competing operations that access the same edge data, and thus eliminates the data race conditions. In this paper, we hereafter refer to the competing operations to the edges as conflicts, and consider two kinds of conflicts: read-write and write-write conflicts.

We further give three methods that can be used by the update functions of a graph algorithm to guarantee the atomicity of its individual read or write:

1) Explicitly locking/unlocking the edge data: by this method, reads or writes to an edge must first acquire the edge’s lock, and after accessing, the lock will be released.

2) Leveraging the support from architecture: in modern computer architecture, the data to be used by a processor need to be first fetched in the cache, and the data written to a memory location will be also first written to the cache. Data are transferred between memory and cache in fixed-size blocks, i.e., cache lines. Since the transfer by cache lines is atomic, update functions can leverage such atomicity in data transfer, provided the accessed data stored in one cache line.

3) Leveraging the support from programming language: in order to prevent synchronization overheads, many programming languages provide the atomic primitives (e.g., C++ [21]) that guarantee the atomic execution of operations. The update function can use this support to guarantee the atomicity of its individual reads and writes.

These methods of implementing atomicity of individual read and write in updates incurs different synchronization overheads during the execution of the algorithms. We will show their corresponding effects to the algorithms’ performance in nondeterministic executions in Section V.

IV. CONVERGENCE IN NONDETERMINISTIC EXECUTION

Considering the read-write conflicts, we give Theorem 1 as the follows.

**Theorem 1.** Provided algorithm \( A \) on graph \( G \) converges with synchronous model execution, and its nondeterministic execution results only read-write conflicts in the edges, algorithm \( A \) will converge on \( G \) when running nondeterministically in our system model.

*Proof:* In order to prove \( A \) converges when running nondeterministically, we need to prove that it takes finite amount of iterations for \( A \) to finish computing with nondeterministic execution.

Assume it spends a total of \( T \) iterations for algorithm \( A \) to converge on graph \( G \) with synchronous model execution. Arbitrarily take a vertex \( v \), whose update function \( f(v) \) repeats for \( k \) times \((k \leq T)\) before \( D_v \) converges to its final result of \( D_v^\ast \). Because the computation advances exactly one step forward by one iteration in synchronous model execution, there must exist a series of vertices denoted by \( v_0, v_1, ..., v_{k-1}, \) and \( v \), forming a chain from \( v_0 \) to \( v \). The computing result is passed from \( v_i \) to \( v_{i+1} \) \((i \in [0, k-2])\) at each iteration, and finally arrives \( v \).

In order to prove that the computing of update function \( f(v) \) converge by nondeterministic execution, we need to prove that it takes finite amount of iterations for the system to pass the computing results from \( v_0 \) to \( v \). The proof can be further reduced as: arbitrarily taking two adjacent vertices, \( v_i \) and \( v_{i+1} \), where \( i \in [0, k-2] \), from the chain, we need to prove that the computing result from \( f(v_i) \) can be passed to \( f(v_{i+1}) \) in finite amount of iterations.

In an iteration, there exist three possible execution orders between \( f(v_i) \) and \( f(v_{i+1}) \) by nondeterministic execution: \( f(v_i) \prec f(v_{i+1}), f(v_i) \parallel f(v_{i+1}) \) and \( f(v_i) \succ f(v_{i+1}) \). We discuss them separately:

1) \( f(v_i) \prec f(v_{i+1}) \): In this case, \( f(v_i) \) executes before \( f(v_{i+1}) \), and \( f(v_{i+1}) \) can use the computing results from \( f(v_i) \). By this way, the results from \( f(v_i) \) can be naturally passed to \( f(v_{i+1}) \), and the passing happens at the same iteration where \( f(v_i) \) is conducted.

2) \( f(v_i) \succ f(v_{i+1}) \): In this case, \( f(v_{i+1}) \) executes before \( f(v_i) \), and thus cannot use the computing results from \( f(v_i) \).
After the execution of \( f(v_i) \), the computing results from \( f(v_i) \) will be written to the objects of its scope. As there exists only read-write conflicts, the updates made by \( f(v_i) \) will be applied to edge \((v_i \rightarrow v_{i+1})\) according to Lemma 1, and will be passed to \( f(v_{i+1}) \) at next iteration.

3) \( f(v_i) \parallel f(v_{i+1}) \): In this case, \( f(v_i) \) and \( f(v_{i+1}) \) conduct computing in parallel. Since only read-write conflicts exist in the execution, the computing result of \( f(v_i) \) will be applied to edge \((v_i \rightarrow v_{i+1})\) according to Lemma 1, and passed to \( f(v_{i+1}) \) at next iteration.

Since vertex \( v \) is arbitrarily taken from \( G \), and the amount of iterations required by \( f(v) \) to converge has been proven to be finite, the amount of iterations needed by the algorithm to converge is also finite.

Fixed point iteration graph algorithms [22], such as PageRank, Sparse Matrix-Vector Multiplication (SpMV) and many others, can use this sufficient condition to guide their nondeterministic executions, since such algorithms generally converge by the synchronous model executions, and only read-write conflicts exist in the edges by pull mode programming. Take PageRank as an example, the update function in PageRank algorithm first reads the input weight values from the incoming edges of its corresponding vertex, and then writes the computed weight value to its out-going edges. Therefore, nondeterministic execution of PageRank results in read-write conflicts on the edges.

However, different from deterministic executions, the execution paths of the updates vary from one run to another in nondeterministic executions, and such variations cast shadows to the final results of fixed point iteration graph algorithms, especially when their convergences are expressed as relative relations between old values and new results of the vertices (i.e., approximation). We will show such phenomenon in the computing results of PageRank in Section V. Moreover, the applicability of Theorem 1 can be extended to graph algorithms that originally converge by a deterministic scheduler of asynchronous model, since similar chain-to-converge also exists.

Considering write-write conflicts, we give Theorem 2 as the follows.

**Theorem 2.** Provided algorithm \( A \) on graph \( G \) converges by deterministic asynchronous model execution, and satisfies monotonicity property, it will converge on graph \( G \) when running nondeterministically even if the nondeterministic execution incurs write-write conflicts in the edges besides the read-write conflicts.

Proof: Suppose there are two arbitrarily taken adjacent vertices \( v \) and \( u \) connected by an edge \((v \rightarrow u)\). At the \( n^{th} \) iteration, both \( f(v) \) and \( f(u) \) are scheduled. The computing result that is produced by \( f(v) \) and going to be written to the edge is denoted as \( D^v_{n \rightarrow u} \), and that from \( f(u) \) is denoted as \( D^u_{n \rightarrow u} \) (\( D^v_{n \rightarrow u} \neq D^u_{n \rightarrow u} \)). By nondeterministic execution, these two update functions will produce write-write conflict on edge \((v \rightarrow u)\).

Since the algorithm converges by asynchronous model deterministic execution, and satisfies monotonicity property (i.e., the computing results monotonically increase or decrease, but not both[23]), there must exist an execution path for \( f(v) \) and \( f(u) \) to converge during the deterministic execution. Without losing generality, let us assume that \( f(v) \rightarrow f(u) \) is the correct execution path for convergence, which means: 1) The data value for convergence at edge \((v \rightarrow u)\) is \( D^v_{n \rightarrow u} \). 2) After receiving \( D^v_{n \rightarrow u} \) from the edge, \( f(u) \) will converge.

We now come back to the nondeterministic execution. Obviously, the best case is that \( f(v) \) wins the competition and writes the correct value of \( D^v_{n \rightarrow u} \) to the edge \((v \rightarrow u)\), and the following execution of \( f(u) \) converges by receiving \( D^v_{n \rightarrow u} \) from the edge. Here we consider the worst case, i.e., \( f(u) \) wins the competition, and at the end of the \( n^{th} \) iteration, \( D^v_{n \rightarrow u} \) will be committed to the (wrong) value of \( D^v_{n \rightarrow u} \) (according to Lemma 2). Since at the \((n+1)^{th}\) iteration, the execution order between \( f(v) \) and \( f(u) \) has three possibilities: \( f(v) \prec f(u) \), \( f(v) \parallel f(u) \) and \( f(v) \succ f(u) \), we divide them into two groups:

1) \( f(v) \prec f(u) \): in this case, \( f(v) \) is scheduled before \( f(u) \), and will update the data of edge \((v \rightarrow u)\) to the correct value of \( D^v_{n \rightarrow u} \). After that, when \( f(u) \) is scheduled, it will read the correct value from the edge, and converge.

2) \( f(v) \parallel f(u) \) or \( f(v) \parallel f(u) \): in these cases, the value read by \( f(u) \) from edge \((v \rightarrow u)\) will be \( D^v_{n \rightarrow u} \). Since the value is what \( f(u) \) think the edge should have, it will falsely converge and not generate new updates to the edge.

However, the execution of \( f(v) \) will make corrections on the edge by changing its value to \( D^v_{n \rightarrow u} \). Therefore, as there is no write-write conflict in the \((n+1)^{th}\) iteration, \( D^v_{n \rightarrow u} \) will committed to \( D^v_{n \rightarrow u} \). After that, \( f(u) \) will truly converge by reading \( D^v_{n \rightarrow u} \) from the edge at the \((n+2)^{th}\) iteration.

By the above discussion, we have proved that the data corruption by write-write conflict can be corrected in finite amount of iterations, and the updates that result in such conflicts also converge in finite amount of iterations. As \( v \) and \( u \) are arbitrarily taken, taking data corruptions and converging in finite amount of iterations apply to all updates that raises write-write conflicts.

Fig. 2 gives an example to explain Theorem 2. The

---

**Figure 2.** An example of write-write conflicts in the edge (the values in brackets denote the intermediate results for a possible iteration).
example is taken from GraphChi’s example implementation of the Weakly Connected Components (WCC) algorithm, we slightly changed it to run nondeterministically. The update function in this example first compares the label values of its corresponding vertex and those of its incident edges, computes the minimal label value, and then updates the label value of its corresponding vertex and its incident edges to the minimal value. When executed nondeterministically, write-write conflicts on the edges will happen.

Assume the initial label values of $v$ and $u$ in Fig. 2 are 1 and 2 respectively, and the initial label value of the edge $(v \rightarrow u)$ is infinite. By the first iteration of nondeterministic execution, the result from $f(u)$ overwrites (corrupts) that from $f(v)$. That is, at the end of the first iteration, $L_{(v \rightarrow u)} = 2$. In the second iteration, $f(v)$ and $f(u)$ overlap with each other for three possibilities: $f(v) \prec f(u)$, $f(v) \parallel f(u)$ and $f(v) \succ f(u)$:

1) $f(v) \prec f(u)$: In this case, $f(v)$ will correct the label value of the edge from 2 to 1, and the following execution of $f(u)$ will correct the label value of $u$ to 1. After that, both $f(v)$ and $f(u)$ converge to correct results in the second iteration.

2) $f(v) \parallel f(u)$ or $f(v) \succ f(u)$: In these cases, the observed label value of the edge from $f(u)$ is 2, and by comparing it with $u$’s label, the computation of $f(u)$ comes to the same result of 2. $f(u)$ thus falsely converges and produces no write to the edge. $f(v)$ will generate the only write in this (the second) iteration to change the label value of the edge to 1. After that, the label value of vertex $u$ will be changed to correct value in the subsequent (the third) iteration. By this way, the write-write conflicts happened on edges are corrected and both $f(v)$ and $f(u)$ converge.

Since the graph traversal algorithms, which are employed to search for chosen structure targets in a graph, generally satisfy monotonicity property, and thus can use the sufficient conditions presented in Theorem 2 to guide their nondeterministic executions. Moreover, as the convergence of such algorithms are generally expressed as absolute conditions, which are not sensitive to the orders of scheduling, their nondeterministic executions will produce the same final results as their deterministic executions.

V. Empirical evaluations

In this section, we evaluate the performances of the nondeterministic executions of representative algorithms on real-world graphs, as well as the results produced by the executions of the selected fixed point iteration algorithm, i.e., PageRank. Subsection V-A presents the experiment setup. Subsection V-B compares the performances. Subsection V-C discusses the variations in the results of PageRank from one run to another.

A. Experiment setup

The test-bed for experiments in this section is a PC server, with two 2.6-GHz Intel Xeon E5-2670 processors, each of which has 8 cores (i.e.,16 cores totally) and 20MB L3 cache. The hyper-threading feature is turned off in all experiments. The server is configured with 64GB of RAM, and one 300GB SAS hard disk drive. 64-bit Redhat enterprise server version 6.2 is installed as the OS, and GCC version 4.8.3 is used to compile the source codes. Four real-world directed graphs as listed in Table I are taken in our experiments: cage15 is taken from the University of Florida Sparse Matrix Collection [24], and the other three graphs come from the Stanford Large Network Dataset Collection [25].

We choose one fixed point iteration graph algorithm, i.e., PageRank, and three graph traversal algorithms, i.e., Weakly Connected Components (WCC), Single Source Shortest Path (SSSP), Bread-First Search (BFS), for evaluations, and choose GraphChi [4] (C++ version 0.2) to execute these algorithms both deterministically and nondeterministically. The reason for choosing GraphChi is that it provides programming interfaces for algorithms’ nondeterministic executions besides the default deterministic executions. Moreover, the memory footprint of the loaded graph in GraphChi is rather small, such that the graphs listed in Table I are fully loaded in memory in all our experiments.

As the source codes of the selected algorithms (as well as the patch to GraphChi) are open for free download1, we briefly introduce their implementations in the following.

PageRank: we implement the algorithm by the concept of local convergence (similar as in [2]). By a small input parameter $\varepsilon$, if the condition of $|f(D_v) - D_v| < \varepsilon$ satisfies, the computation on vertex $v$ converges. Each vertex stores an initial float type weight value of 1 and each edge also stores a float type weight value, whose initial value is 1 divided by the out-degree of the vertex. The update function will read in all weight values of the incoming edges, add them to the weight value of its corresponding vertex, and then divide the summation by the out-degree. The weight values of the out-going edges are finally updated by the quotient from the division. During nondeterministic executions, there will be read-write conflicts happening on the edges.

WCC: the example program shipped together with GraphChi’s source code is used for deterministic execution, and it is just slightly changed (to choose nondeterministic scheduler) for nondeterministic execution. As discussed in Section IV, during nondeterministic executions, this example algorithm will result in write-write conflicts in the edges.

<table>
<thead>
<tr>
<th>Graphs</th>
<th>$V$</th>
<th>$E$</th>
<th>Raw file size</th>
</tr>
</thead>
<tbody>
<tr>
<td>web-BerkStan</td>
<td>665,231</td>
<td>7,600,595</td>
<td>106MB</td>
</tr>
<tr>
<td>web-Google</td>
<td>916,428</td>
<td>5,105,040</td>
<td>111MB</td>
</tr>
<tr>
<td>soc-LiveJournal1</td>
<td>4,847,570</td>
<td>68,993,773</td>
<td>1.6GB</td>
</tr>
<tr>
<td>cage15</td>
<td>5,154,859</td>
<td>94,044,692</td>
<td>2.2GB</td>
</tr>
</tbody>
</table>

1https://github.com/mrshawcode/GraphChi_nondeter_algorithm
SSSP: in this algorithm, each vertex stores a distance value to denote the distance from the single source. Initially, the distance value is set to be 0 for the single source, and infinity for other vertices. Each edge stores a initial fixed weight value, which is a random value (between 1 and 10) generated during initialization, and a distance value, which is initially set to be the same as the distance value of its source vertex. The updates pass the computing results via the edges, and when executing nondeterministically, only read-write conflicts happen in the edges.

BFS: this algorithm can be regarded as a special case of SSSP, where the weigh values of the edges are all ones. And similar as SSSP, the nondeterministic executions of this algorithm result in read-write conflicts that happen in the edges.

All these algorithms originally converge by the default deterministic scheduler of GraphChi, and thus converge after being modified to run nondeterministically as proven in Theorem 1 and 2. We apply all three methods that implement the atomicity of individual read and write to these algorithms for nondeterministic executions: 1) By the explicit locking/unlocking method, a lock is defined for each edge, and an access to the edge must first acquire the lock and release the lock when finished accessing. 2) By the method of leveraging the architecture support, as the data structure for the edges in above algorithms are rather small, we align the edge data structures of the above algorithms to 8 bytes (i.e., the width of data bus), such that they are stored in single cache line of our test-bed (i.e., 64 bytes). 3) By the method of leveraging the compiler support, we choose the relaxed atomic primitives of C++ (i.e., memory_order_relaxed) to implement the atomicity of the reads and writes in the update functions.

B. Performance comparisons

Fig. 3 illustrates the computing times (note we excluded the I/O times that are spent on graph loading) of the graph algorithms by both deterministic and nondeterministic executions. Since the performances of the algorithms by the built-in external deterministic scheduler in GraphChi does not scale (the updates are actually conducted sequentially due to the data dependences among the updates), we only show those with four threads.

The first notable fact that can be observed from Fig. 3 is that by leveraging the architecture support, the algorithms achieve best performances by nondeterministic executions, while the method of explicit locking/unlocking largely degrades the performance due to the synchronization overheads. By leveraging the architecture support, the reduction on execution time by nondeterministic execution in our testbed can be up to about 70% (see WCC on web-BerkStan with 8 threads), i.e., about 3.3x performance boost. The method of leveraging compiler support results marginally worse performances than those of using architecture support. The reason is that the compiler automatically chooses the method that incurs smallest synchronization overhead to implement the atomicity of operations by the relaxed atomic primitives. This method should be similar or close to that of architecture support.

The second notable fact that can be observed from Fig. 3 is that by all the three methods that implement atomicity of individual read and write, the performances of algorithms by nondeterministic executions scale with the number of available threads, and the scaling exhibits similar patterns with a few exceptions (e.g., those on web-Google). However, it can also be observed that the scaling is not linear. This is because graph-mining algorithms are typical memory-intensive applications and have bad locality on memory accessing during execution. When the number of threads increases, the bandwidth between processors and memory will be gradually saturated, which counteracts the performance gains from parallelism.

Another interesting fact that can be observed from Fig. 3 is that in some cases (e.g., BFS and SSSP on cage15, all algorithms on web-Google), the performances of algorithms by nondeterministic executions with explicit locking/unlocking are even better than those of the original deterministic executions when giving enough (e.g., 16 in such cases) processing cores. This fact reveals the power of nondeterministic execution even with a suboptimal design of synchronization.

C. Variations in the results of PageRank

As there are no predefined scheduling orders for the updates in nondeterministic executions of the graph algorithms, the computed results of fixed point iteration graph algorithms at convergence can be different from one run to another. In this subsection, we will show such variations in the results of PageRank, and discuss how it is affected by environment of execution and convergence conditions. Further discussions (e.g., precision and range of errors of the results) are out of scope and placed to the future works of this paper.

We first run the algorithm deterministically on 4 processing cores (denoted as 4DE) for 5 times, and obtain 5 independent results on the ranking of pages (which are modeled as vertices of web-Google). After that, we execute the algorithm nondeterministically on 4 processing cores (4NE), 8 processing cores (8NE) and 16 processing cores (16NE) respectively, and obtain 5 independent results for each of these scenarios. Varying the processing cores for deterministic execution is meaningless, since by the external deterministic scheduler of GraphChi, the update tasks are conducted by following a predefined (sequential) order. However, varying the number of processing cores will affect the execution order of update tasks by nondeterministic scheduler, and thus affect the produced results.

To compare the difference between two independent computing results, we compute the minimal index where the
two results differ, and name it as difference degree. For example, suppose we have two results \( r_1 \) and \( r_2 \), where \( r_1 = \{1, 2, 3, 5, 7\} \), and \( r_2 = \{1, 2, 3, 7, 5\} \). Assume the suffix of the result array begins from 0, the difference degree by comparing \( r_1 \) and \( r_2 \) is 3. For PageRank, bigger difference degree means that the variation happens in the page of less significance, i.e., bigger is better.

Table II presents the average difference degrees of the results from the same configurations. Since we have 5 results for each configuration, each figure in Table II is the average of 10 (i.e., \( C^2_5 \)) difference degrees. Table III presents the average difference degrees of results between different configurations, where the figures are computed by averaging.

<table>
<thead>
<tr>
<th>Configuration Comparison</th>
<th>( \varepsilon = 0.1 )</th>
<th>( \varepsilon = 0.01 )</th>
<th>( \varepsilon = 0.001 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>4DE vs. 4DE</td>
<td>20228</td>
<td>10314</td>
<td>5998</td>
</tr>
<tr>
<td>4NE vs. 4NE</td>
<td>294</td>
<td>537</td>
<td>3356</td>
</tr>
<tr>
<td>8NE vs. 8NE</td>
<td>138</td>
<td>328</td>
<td>1289</td>
</tr>
<tr>
<td>16NE vs. 16NE</td>
<td>102</td>
<td>141</td>
<td>409</td>
</tr>
</tbody>
</table>
the difference degrees pairwise, i.e., 5 of them. Note that differences between results from deterministic executions are resulted by the precision limit of float data type.

From Table II, it can be observed that the difference degree of nondeterministic executions are rather small compared with the deterministic executions, which means that the variation happens on more significant pages. And with the increased precision on convergence condition (i.e., smaller $\varepsilon$), the difference degree will increase (i.e., variations happens on less significant pages). Another observation that can be made from Table II is that with increased processing cores, variations in results happen on more significant pages.

This is because by nondeterministic execution, the execution orders of updates are more likely to be affected by the noises (e.g., uncertainty on scheduling, random IRQs, memory stalls) of the environment. Giving more processing units further complicates the environment and introduce more noises. However, with increased precision on convergence condition (i.e., smaller $\varepsilon$), the variations in results will move to less significant pages, since more noises will be filtered by the increased precision.

From Table III, it can be observed that with increased precision on convergence conditions, the variations in results between executions in different environments will move to less significant pages, which is similar phenomenon as observed in Table II. Combining the data from both Table II and Table III, it can be observed that for the pages with higher rank (e.g., ranking number smaller than 100), the results from all these selected scenarios are identical. As there is no evidence showing that divergence on lower rank pages (e.g., ranking number bigger than 100) affects practical use, the data of Table II and Table III actually underline the usability of nondeterministic execution of PageRank from another angle. However, this conclusion may not apply to other fixed point iteration algorithms.

### VI. Related works

Most of existing research works on graph processing by coordinated scheduling of asynchronous model are focused on the deterministic schedulers. For example, the set scheduler in GraphLab [2], the chromatic scheduler in [10], the Deterministic Interference Graph (DIG) scheduler in Galois [11] and the external deterministic scheduler in GraphChi [4]. The common technique that implements the deterministic executions for the graph algorithms in these deterministic schedulers is to carefully arrange the execution paths of the updates, such that the concurrently executed updates have no data dependences among themselves. Such technique, however, incurs huge time and space overheads, and results in less scalable performance for algorithms than the nondeterministic executions. Interestingly, performance results in [11], especially those comparing the performance of deterministic and nondeterministic executions of graph algorithms, endorsed such shortcomings of deterministic executions and confirmed our observations in this paper.

On the other hand, most of the graph processing frameworks that employ the coordinated scheduling of asynchronous model provide programming interfaces for nondeterministic executions of the graph algorithms. Such frameworks include GraphLab, GraphChi, Galois and some others. However, a key ring, which tells whether a graph algorithm is eligible for nondeterministic executions, is missing. Our research works presented in this paper is motivated to give answers to this crucial question.

### VII. Conclusions and future works

In this paper, we show that by guaranteeing the atomicity of individual reads and writes, most existing graph algorithms (i.e., fixed point iteration algorithms and graph traversal algorithms) can be executed nondeterministically in the graph processing frameworks that employ the synchronous implementation of asynchronous model, and achieve scalable performance. Our research opens the doors of more research topics, which are the future works of this paper, as listed below:

- More sufficient conditions (e.g., those considering the push mode), which can be used by the programmers, pertaining to the graph algorithms’ convergence by nondeterministic executions.
- More discussions (e.g., on precision, range of errors) on the variations in the results of fixed point iteration algorithms by nondeterministic executions.
- Theoretical analyses of the convergence speed (e.g., in amount of iterations) of graph algorithms by nondeterministic executions.
- Extending the applicability of results in this paper to more scenarios, such as pure asynchronous model, and distributed systems, by relaxing the system model discussed in this paper.

### Acknowledgments

This work is supported by Natural Science Foundation of China key project under grant No. 61433019.

### References


